TCP, UDP
Model Client – Server
Socket API
Biblioteka Boost::Asio

Michał Filek

20 kwietnia 2016

Zaawansowane techniki programowania

Plan prezentacji:

1.*Warstwy sieci model OSI i TCP/IP*

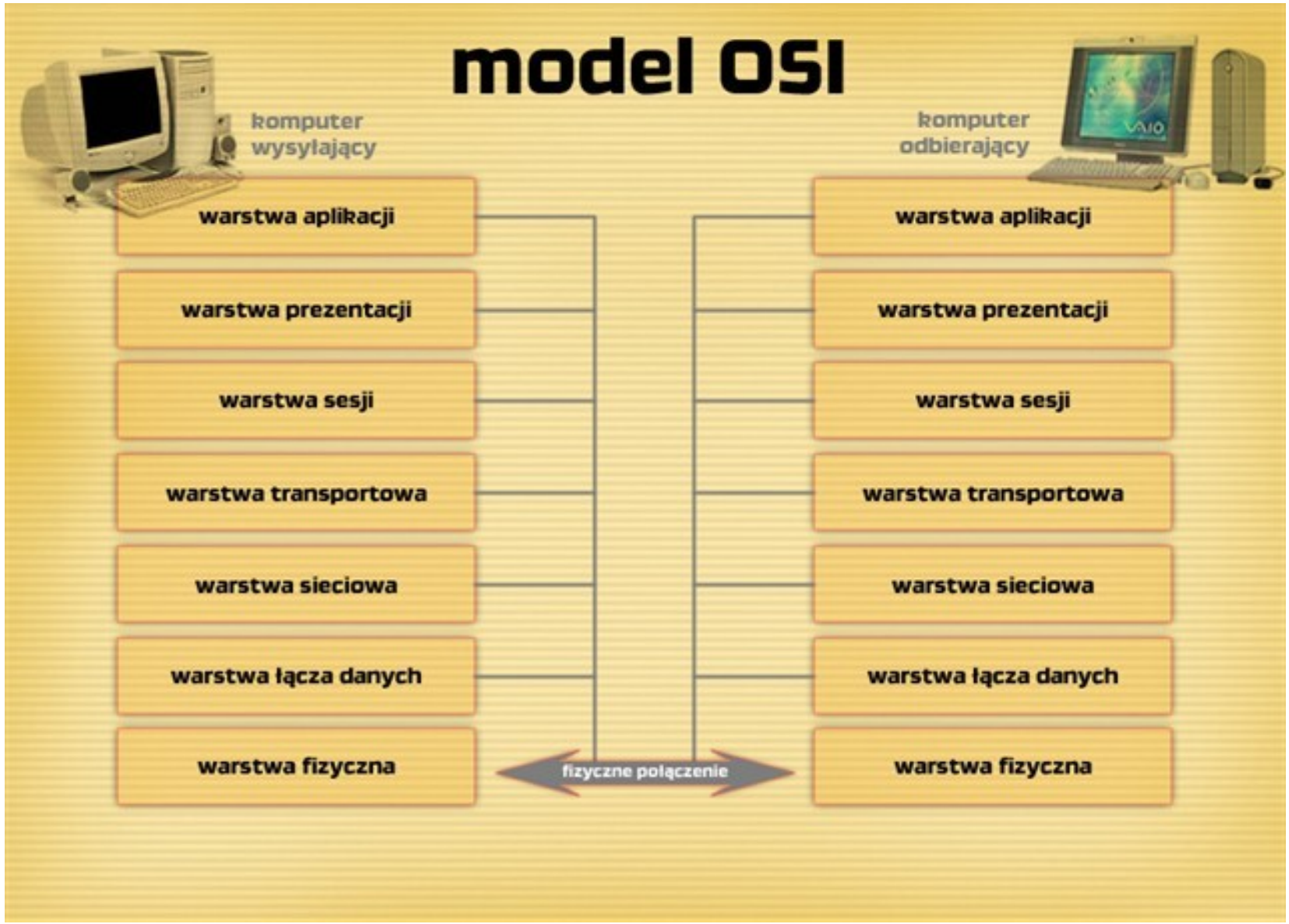
2.*Warstwa transportu: Zestawienie TCP / UDP*

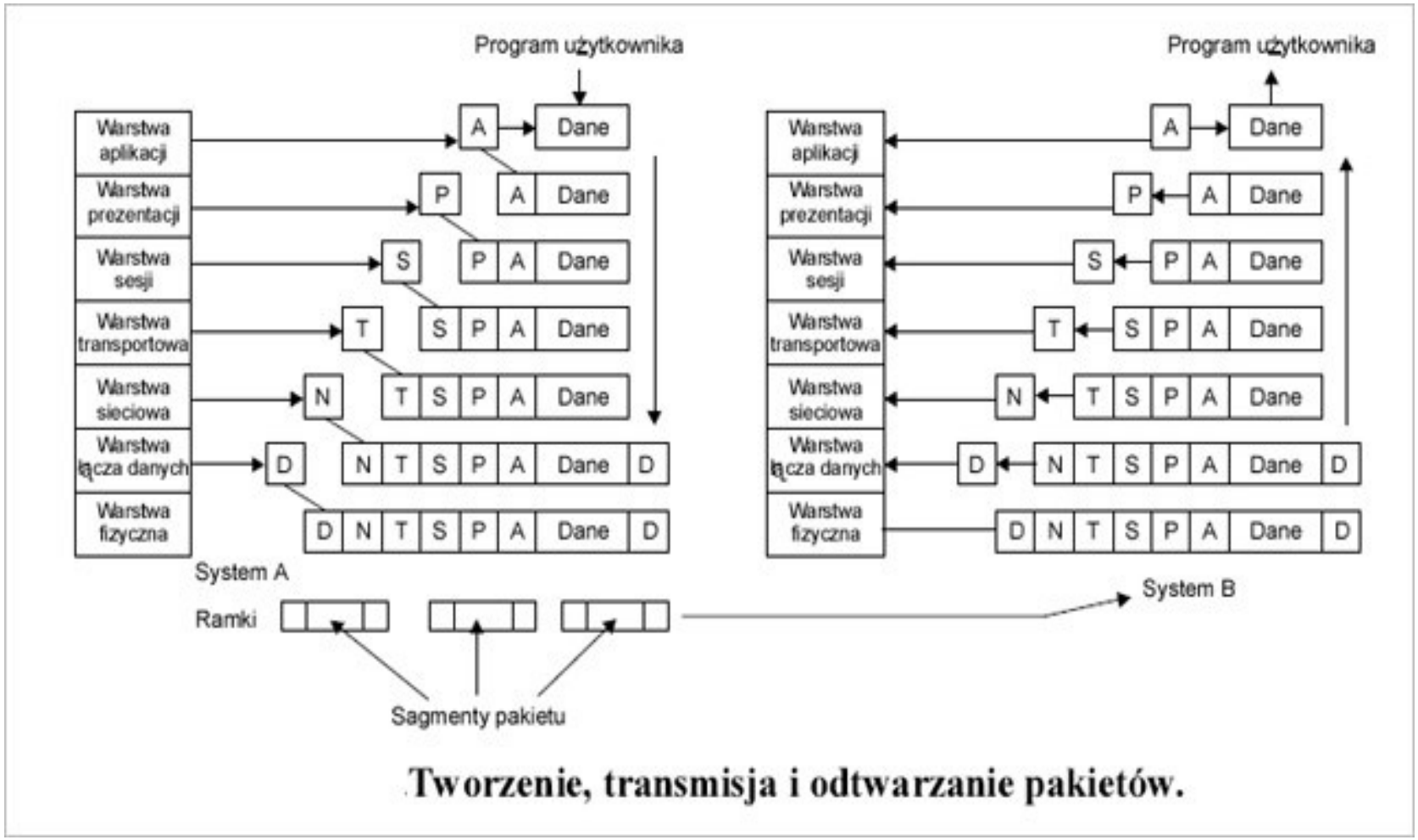
3.*Model sieciowy Client-Serwer*

4.*Socket API*

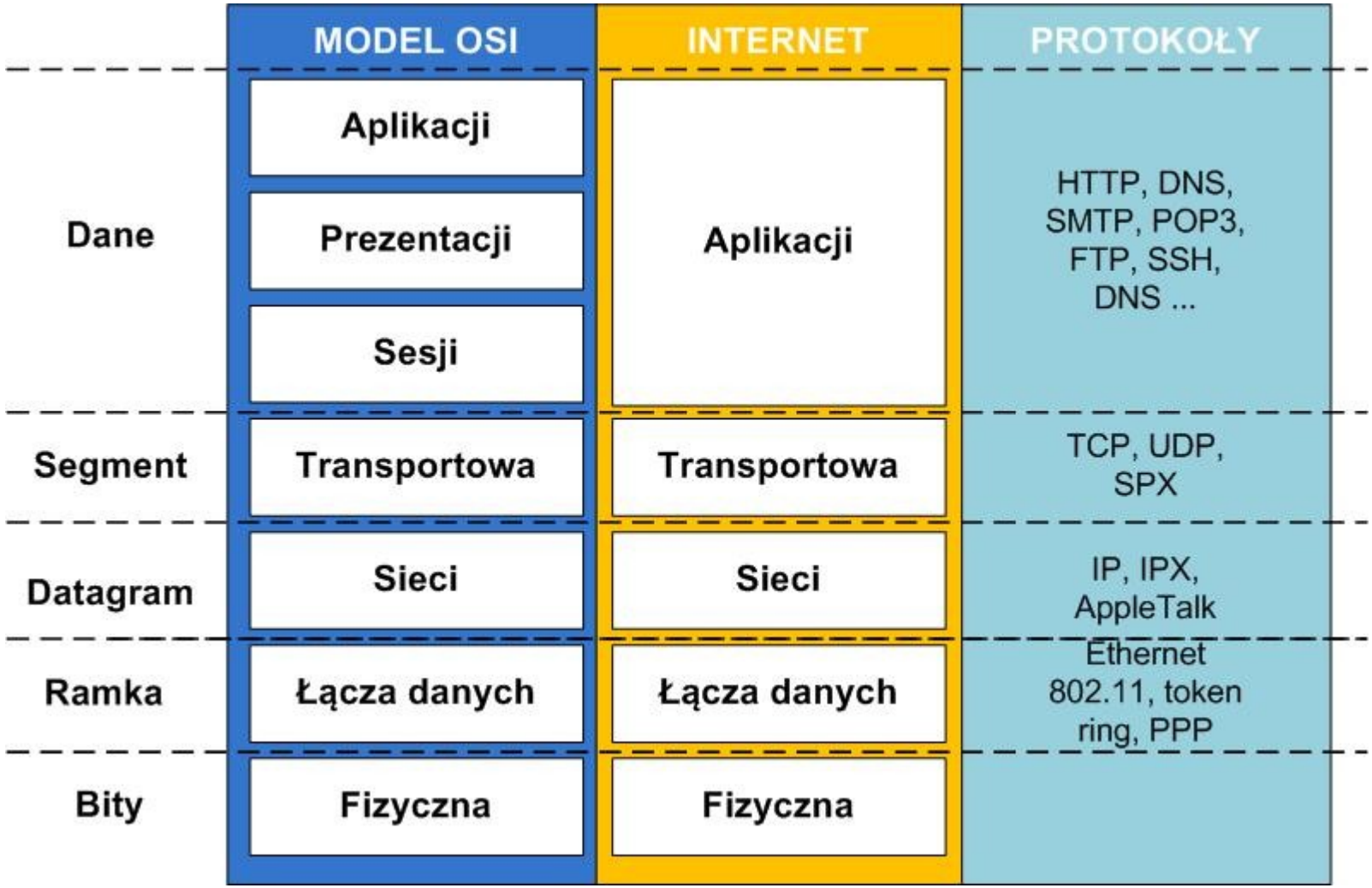
5.*Komunikacja synchroniczna i asynchroniczna*

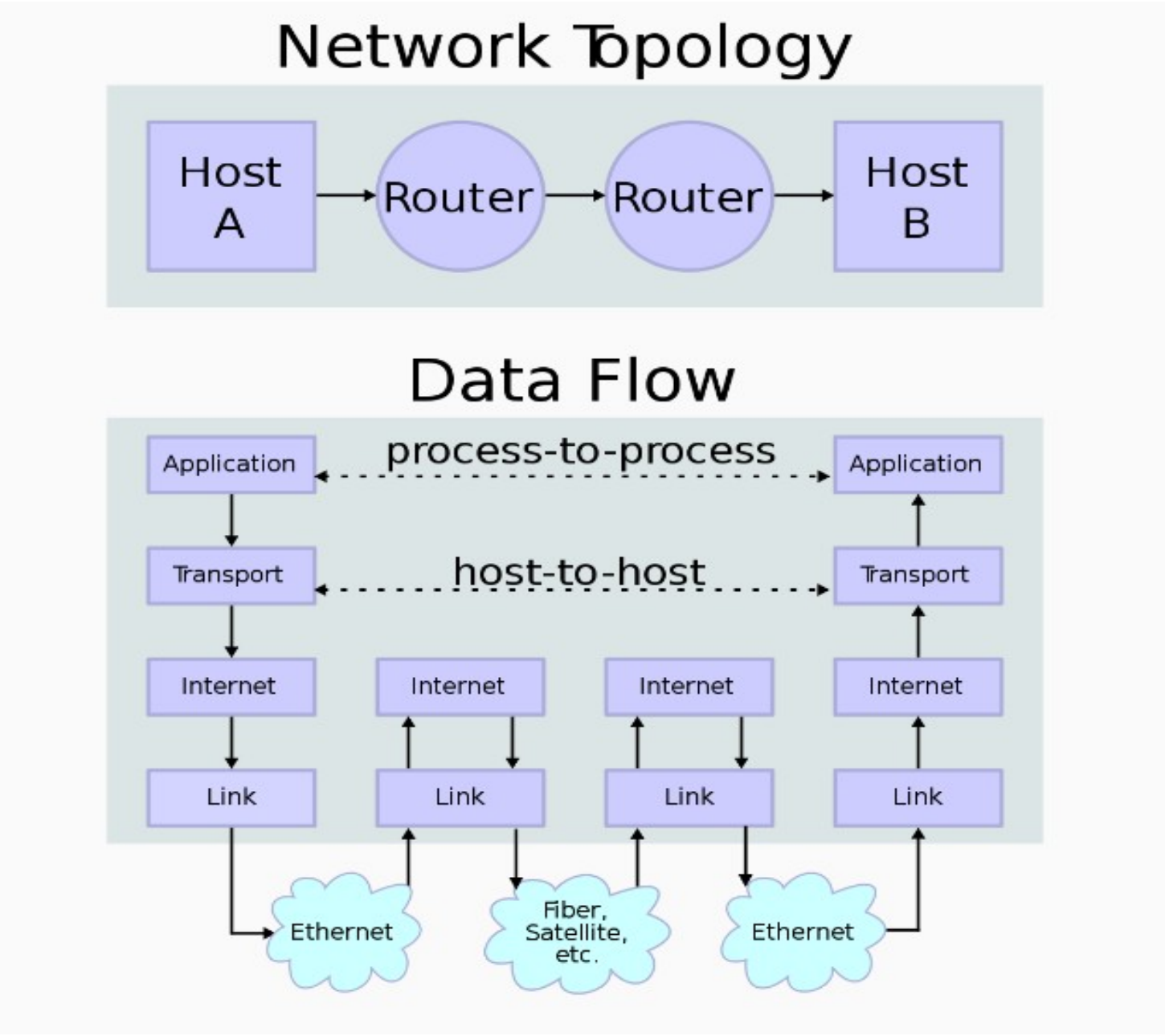
6.*Implementacja prostych aplikacji za pomocą Boost::Asio*


model OSI
komputer wysyłający
komputer odbierający
warstwa aplikacji
warstwa prezentacji
warstwa sesji
warstwa transportowa
warstwa sieciowa
warstwa łącza danych
warstwa fizyczna
fizyczne połączenie
warstwa aplikacji
warstwa prezentacji
warstwa sesji
warstwa transportowa
warstwa sieciowa
warstwa łącza danych
warstwa fizyczna



Tworzenie, transmisja i odtwarzanie pakietów.


MODEL OSI
INTERNET
PROTOKOŁY
Aplikacji
Prezentacji
Sesji
Aplikacji
HTTP, DNS, SMTP, POP3, FTP, SSH, DNS ...
Dane
Segment
Transportowa
Transportowa
TCP, UDP, SPX
Datagram
Sieci
Sieci
IP, IPX, AppleTalk
Ramka
Łącza danych
Łącza danych
Ethernet 802.11, token ring, PPP
Bity
Fizyczna
Fizyczna


Network Topology
Host A
Router
Router
Host B
Data Flow
Application
process-to-process
Application
Transport
host-to-host
Transport
Internet
Internet
Internet
Internet
Link
Link
Link
Link
Ethernet
Fiber, Satellite, etc.
Ethernet

AGH

Warstwa Transportu
Protokoły: TCP UDP

Warstwa transportowa:
- segmentuje dane oraz składa je w tzw. strumień

- zapewnia całościowe połączenie między stacjami: źródłową oraz docelową, które obejmuje całą drogę transmisji

- podział danych na części, które są kolejno indeksowane i wysyłane do docelowej stacji

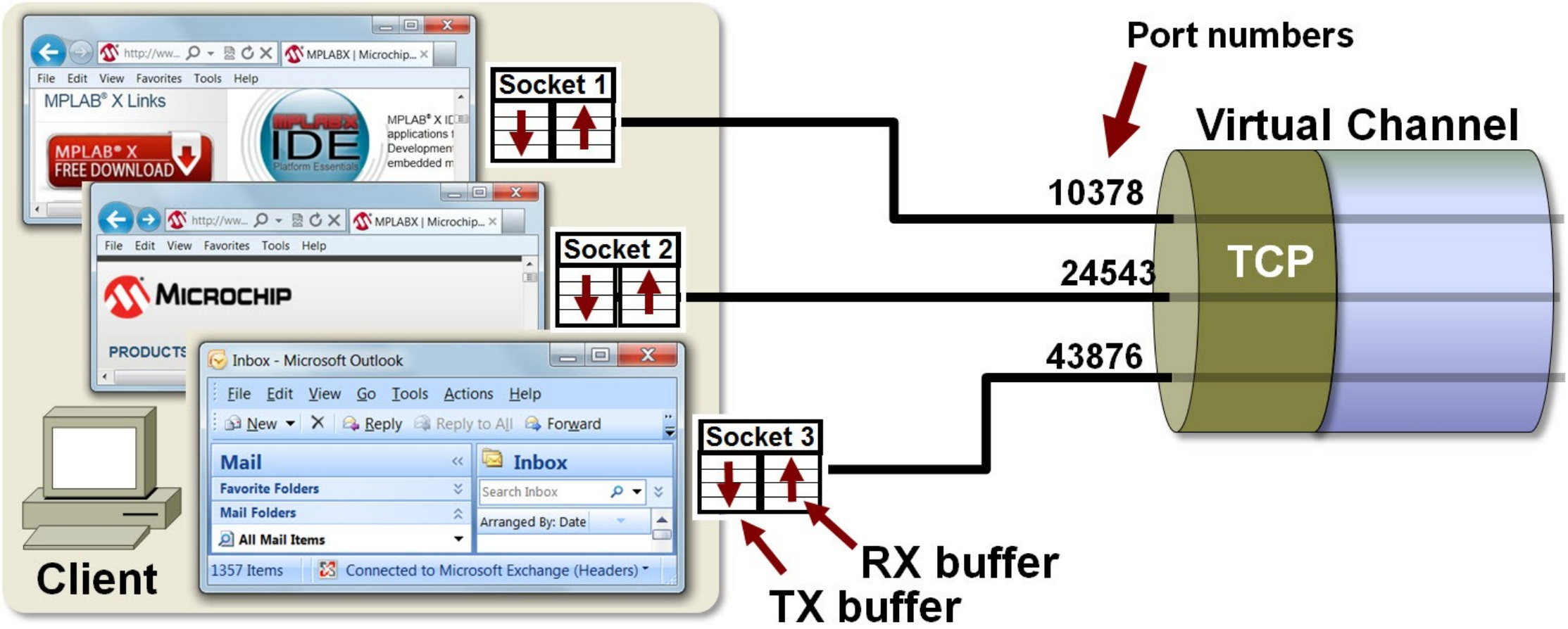
-komunikacja przebiega za pomocą portów.

/*etv/services */

Port protokołu – pojęcie związane z protokołami używanymi w Internecie do identyfikowania procesów działających na odległych systemach. Jest to jeden z parametrów gniazda.

Numery portów reprezentowane są przez liczby naturalne z zakresu od 0 do 65535 (2^{16}-1).
Niektóre numery portów (od 0 do 1023) są określone jako ogólnie znane (ang. well known ports) oraz zarezerwowane na standardowo przypisane do nich usługi, takie jak np. WWW czy poczta elektroniczna.

Dzięki temu można identyfikować nie tylko procesy, ale ogólnie znane usługi działające na odległych systemach.


Port numbers
Virtual Channel
Socket 1
Socket 2
Socket 3
10378
24543
43876
TCP
RX buffer
TX buffer
Client
Inbox - Microsoft Outlook
MPLAB® X Links
MICROCHIP

User Datagram Protocol (UDP):

- bezpołączeniowy
- brak narzutu na nawiązywanie połączenia i śledzenie sesji
- brak kontroli przepływu i retransmisji

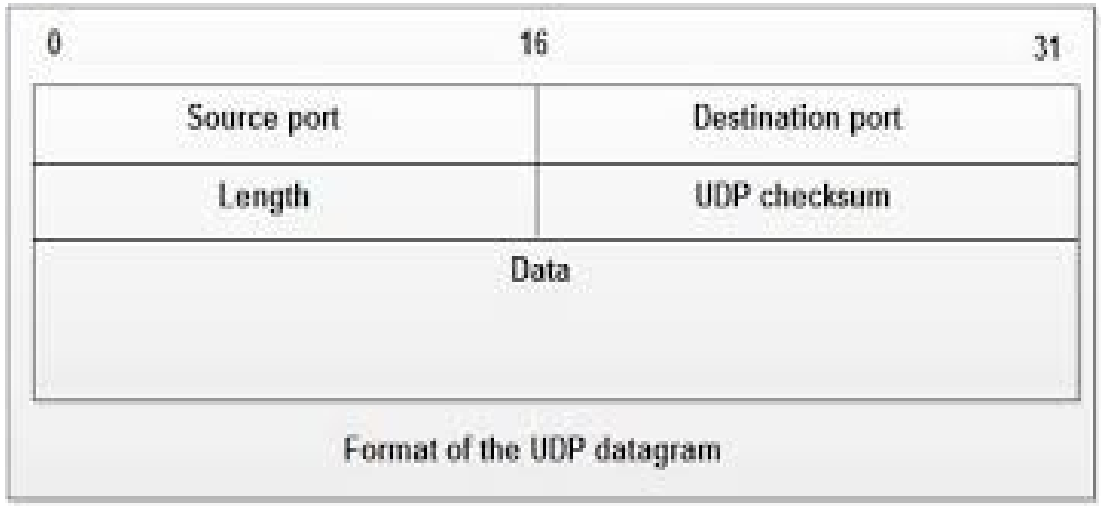
Korzyścią płynącą z takiego uproszczenia budowy jest większa szybkość transmisji danych i brak dodatkowych zadań, którymi musi zajmować się host posługujący się tym protokołem.

Z tych względów UDP jest często używany w takich zastosowaniach jak:

- wideokonferencje

- strumienie dźwięku w Internecie i gry sieciowe, gdzie dane muszą być przesyłane możliwie szybko, a poprawianiem błędów zajmują się inne warstwy modelu OSI.

- protokoly wyzszych warstw uzywajace UDP VoIP lub protokół DNS.



Format of the UDP datagram

Długość

16-bitowe pola specyfikują długość w bajtach całego datagramu: nagłówek i dane.

Suma kontrolna

16 bitowe pole, które jest użyte do sprawdzania poprawności nagłówka oraz danych. Ponieważ IP nie wylicza sumy kontrolnej dla danych, suma kontrolna UDP jest jedyną gwarancją, że dane nie zostały uszkodzone.

Transmission Control Protocol (TCP) :

- połączeniowy
- niezawodny
- strumieniowy protokół komunikacyjny

Wykorzystywany do przesyłania danych pomiędzy procesami uruchomionymi na różnych maszynach.

Opracowano go na podstawie badań Vintona Cerfa oraz Roberta Kahna. Został opisany w dokumencie RFC793.

TCP jest protokołem działającym w trybie klient-serwer. Serwer oczekuje na nawiązanie połączenia na określonym porcie. Klient inicjuje połączenie do serwera.

W przeciwieństwie do UDP, TCP gwarantuje wyższym warstwom komunikacyjnym:

- dostarczenie wszystkich pakietów w całości
- z zachowaniem kolejności i bez duplikatów

Zapewnia to wiarygodne połączenie kosztem większego narzutu w postaci nagłówka i większej liczby przesyłanych pakietów.

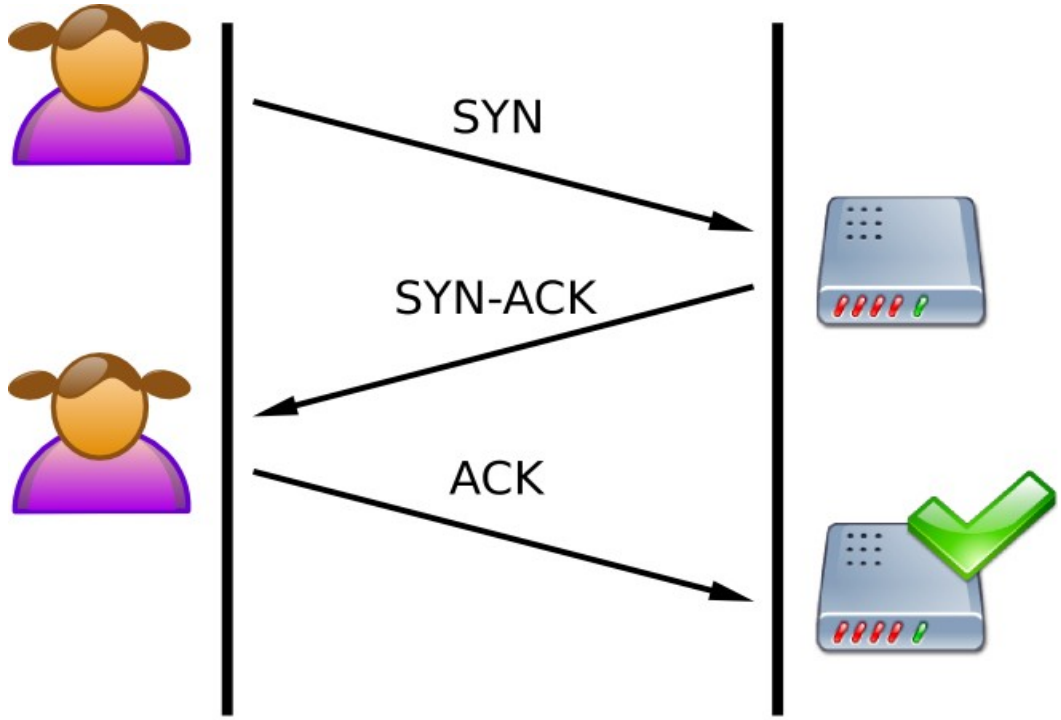
Aplikacje, w których zalety TCP przeważają nad wadami (większy koszt związany z utrzymaniem sesji TCP przez stos sieciowy), to m.in. programy używające protokołów warstwy aplikacji:

- HTTP
- SSH
- FTP
- SMTP/POP3 i IMAP4.

Np. czaty, poczta elektroniczna

Mechanizmy TCP:

1. Three-way handshake (zarowno inicjowanie i kończenie połączenia)



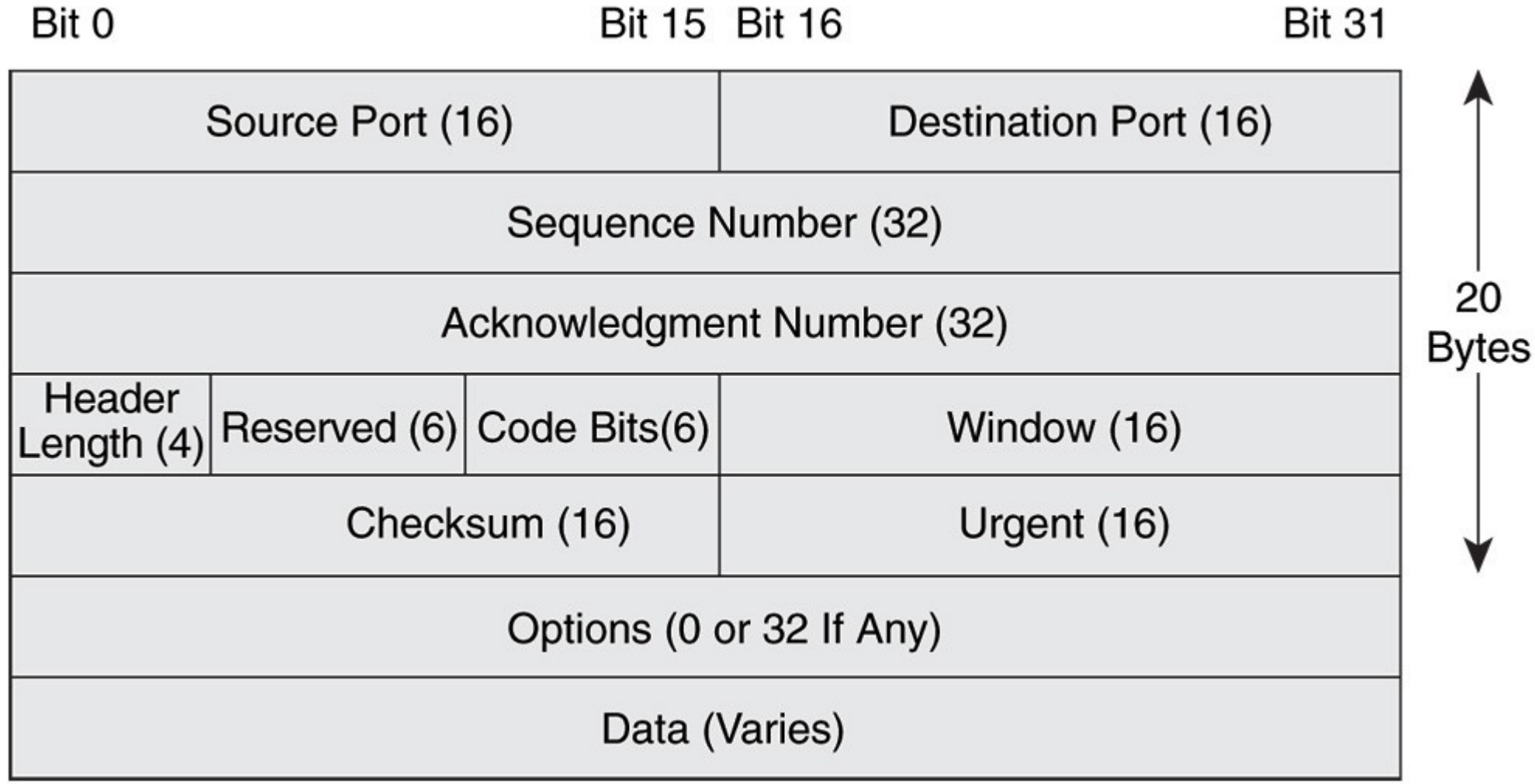
Transmisja danych

- w celu weryfikacji wysyłki i odbioru TCP wykorzystuje sumy kontrolne i numery sekwencyjne pakietów.

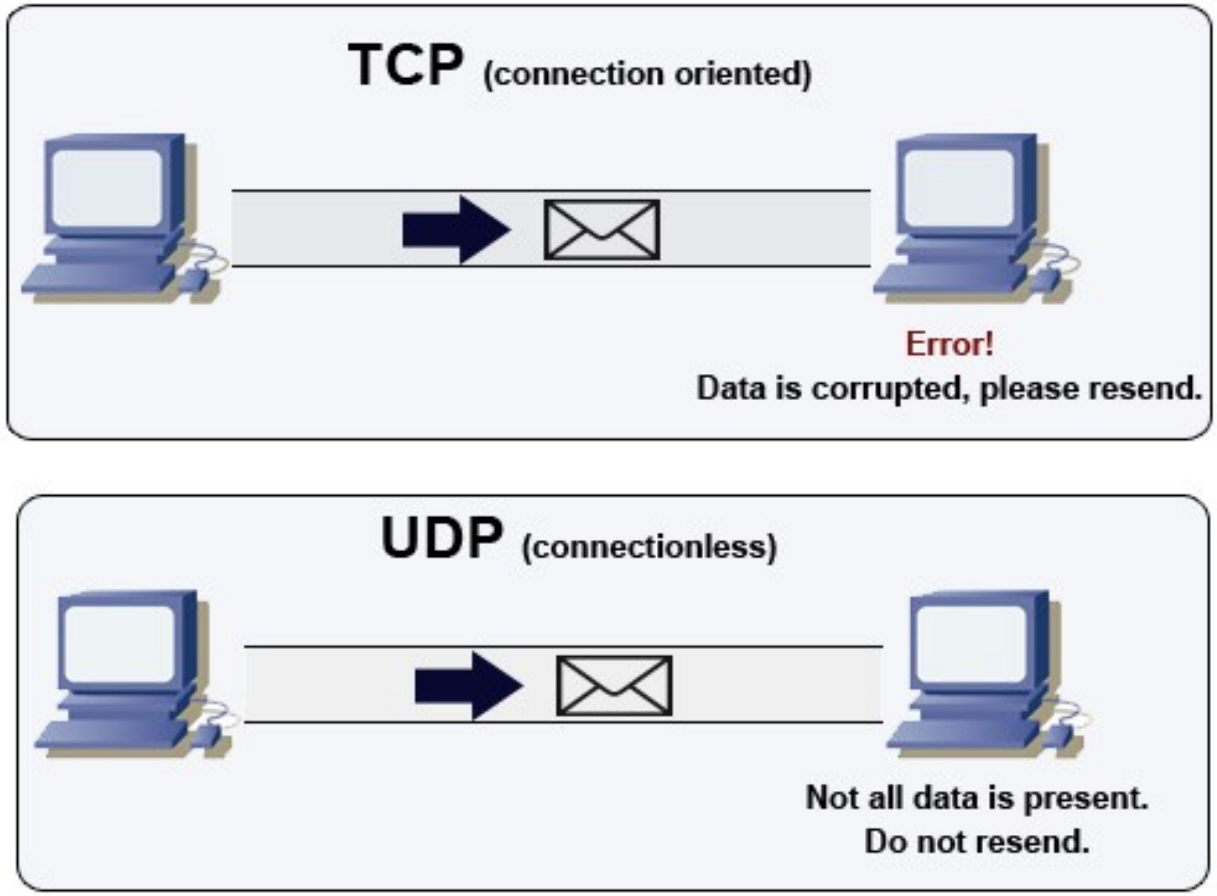
- odbiorca potwierdza otrzymanie pakietów o określonych numerach sekwencyjnych ustawiając flagę ACK.

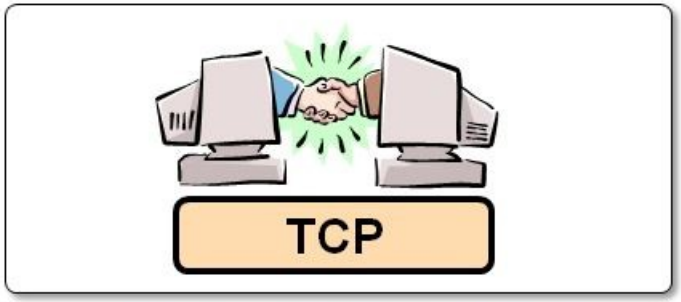
- brakujące pakiety są retransmitowane.

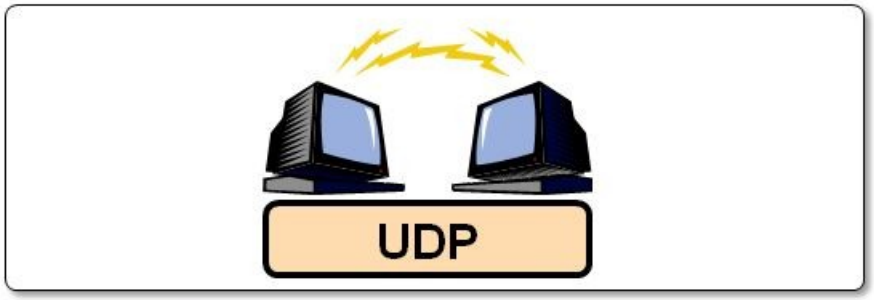
- host odbierający pakiety TCP defragmentuje je i porządkuje je według numerów sekwencyjnych tak, by przekazać wyższym warstwom modelu OSI pełen złożony segment.


Bit 0
Bit 15
Bit 16
Bit 31
Source Port (16)
Destination Port (16)
Sequence Number (32)
Acknowledgment Number (32)
Header Length (4)
Reserved (6)
Code Bits(6)
Window (16)
Checksum (16)
Urgent (16)
Options (0 or 32 If Any)
Data (Varies)
20 Bytes

Zestawienie: TCP a UDP

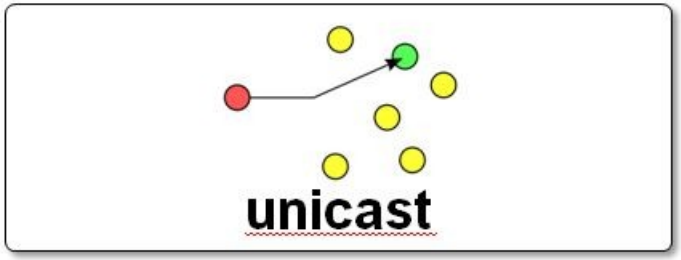


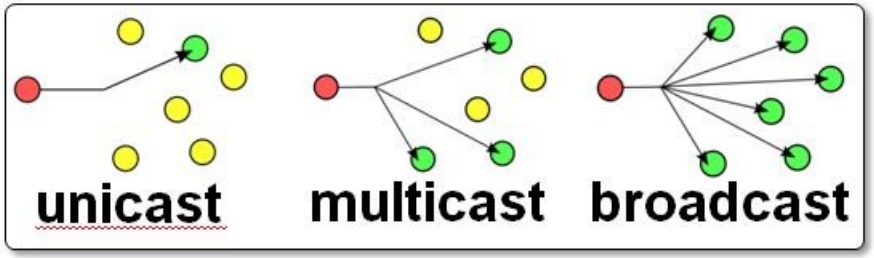




- Slower but reliable transfers
- Typical applications:
  - Email
  - Web browsing

- Fast but non-guaranteed transfers ("best effort")
- Typical applications:
  - VoIP
  - Music streaming



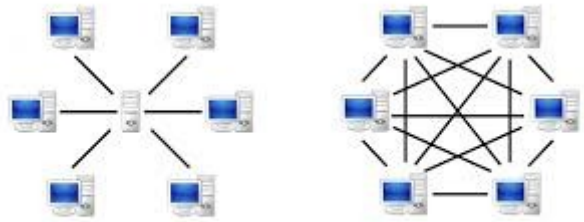


AGH

Model Sieciowy Client -Server

Pierwotnie idea Internetu była zbliżona do sieci P2P – wszystkie hosty pełniły równorzędną rolę w procesie wymiany danych. W wyniku gwałtownego rozwoju, symetria Internetu została złamana.

Rolę dystrybutorów przejęły przedsiębiorstwa i instytucje, które było stać na utrzymanie stałych łącz o bardzo dużej przepustowości i zakup silnych komputerów, zdolnych obsługiwać ruch o dużym natężeniu.

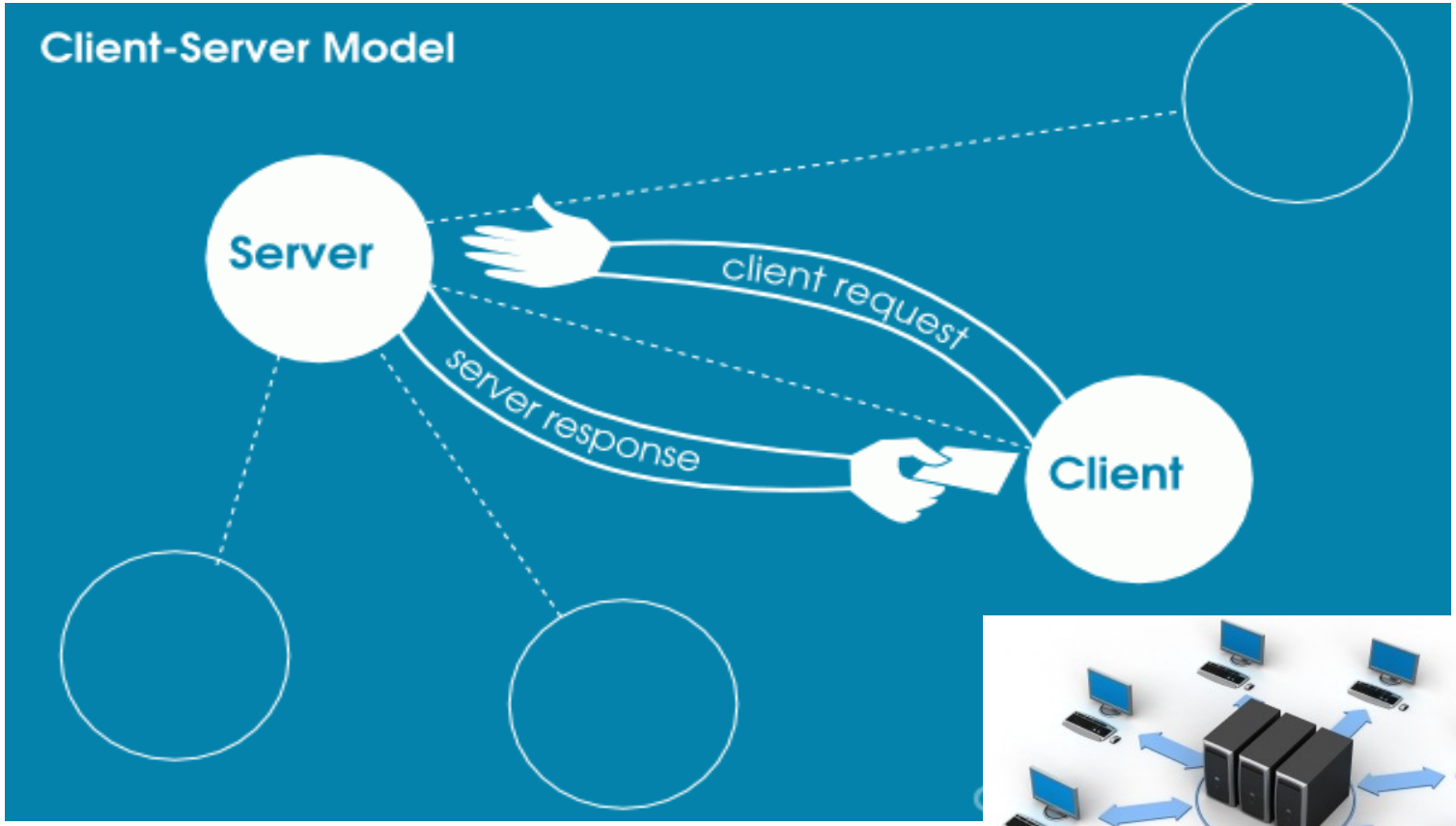


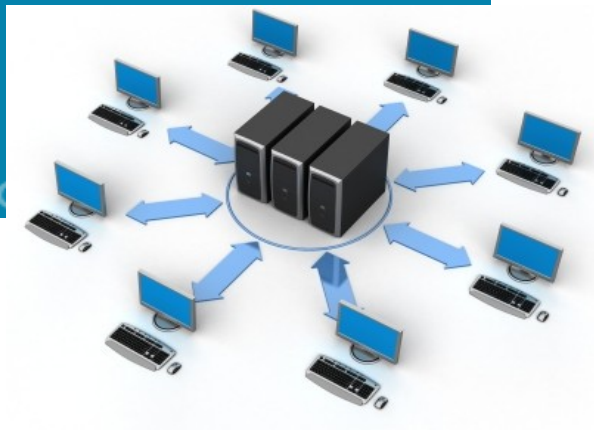
Klient-serwer – architektura systemu komputerowego. Polega to na ustaleniu, że serwer zapewnia usługi dla klientów, zgłaszających do serwera żądania obsługi

Najczęściej spotykane serwery działające w oparciu o architekturę klient-serwer to:

- serwer poczty elektronicznej
- serwer WWW
- serwer aplikacji.

Z usług jednego serwera może zazwyczaj korzystać wiele klientów.


Client-Server Model
Server
client request
server response
Client



Strona klienta żąda dostępu do danej usługi lub zasobu.

Tryb pracy klienta:
aktywny,
wysyła żądanie do serwera,
oczekuje na odpowiedź od serwera.

Strona serwera świadczy usługę lub udostępnia zasoby.

Tryb pracy serwera:
pasywny,
czeka na żądania od klientów,
w momencie otrzymania żądania,
przetwarza je, a następnie wysyła odpowiedź.

AGH

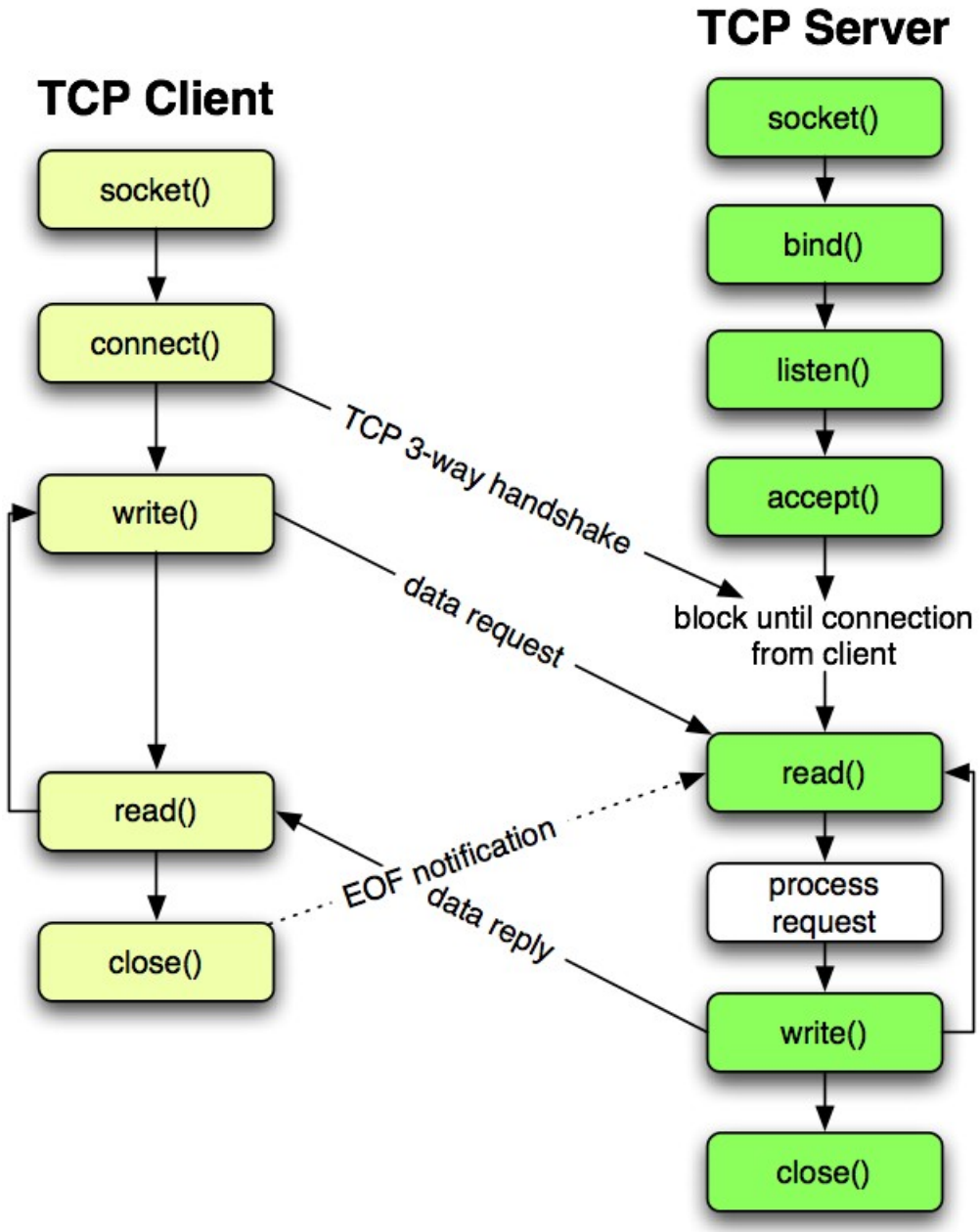
Socket API

Socket(endpoint) – pojęcie abstrakcyjne reprezentujące dwukierunkowy punkt końcowy połączenia. Dwukierunkowość oznacza możliwość wysyłania i odbierania danych.

Gniazdo posiada trzy główne właściwości:

- typ gniazda identyfikujący protokół wymiany danych
- lokalny adres (np. adres IP, IPX, czy Ethernet)
- lokalny numer portu identyfikujący proces, który wymienia dane przez gniazdo

Adres IP wyznacza węzeł w sieci, numer portu określa proces w węźle, a typ gniazda determinuje sposób wymiany danych.


TCP Client
TCP Server
socket()
connect()
write()
read()
close()
socket()
bind()
listen()
accept()
block until connection from client
read()
process request
write()
close()
TCP 3-way handshake
data request
EOF notification
data reply

AGH

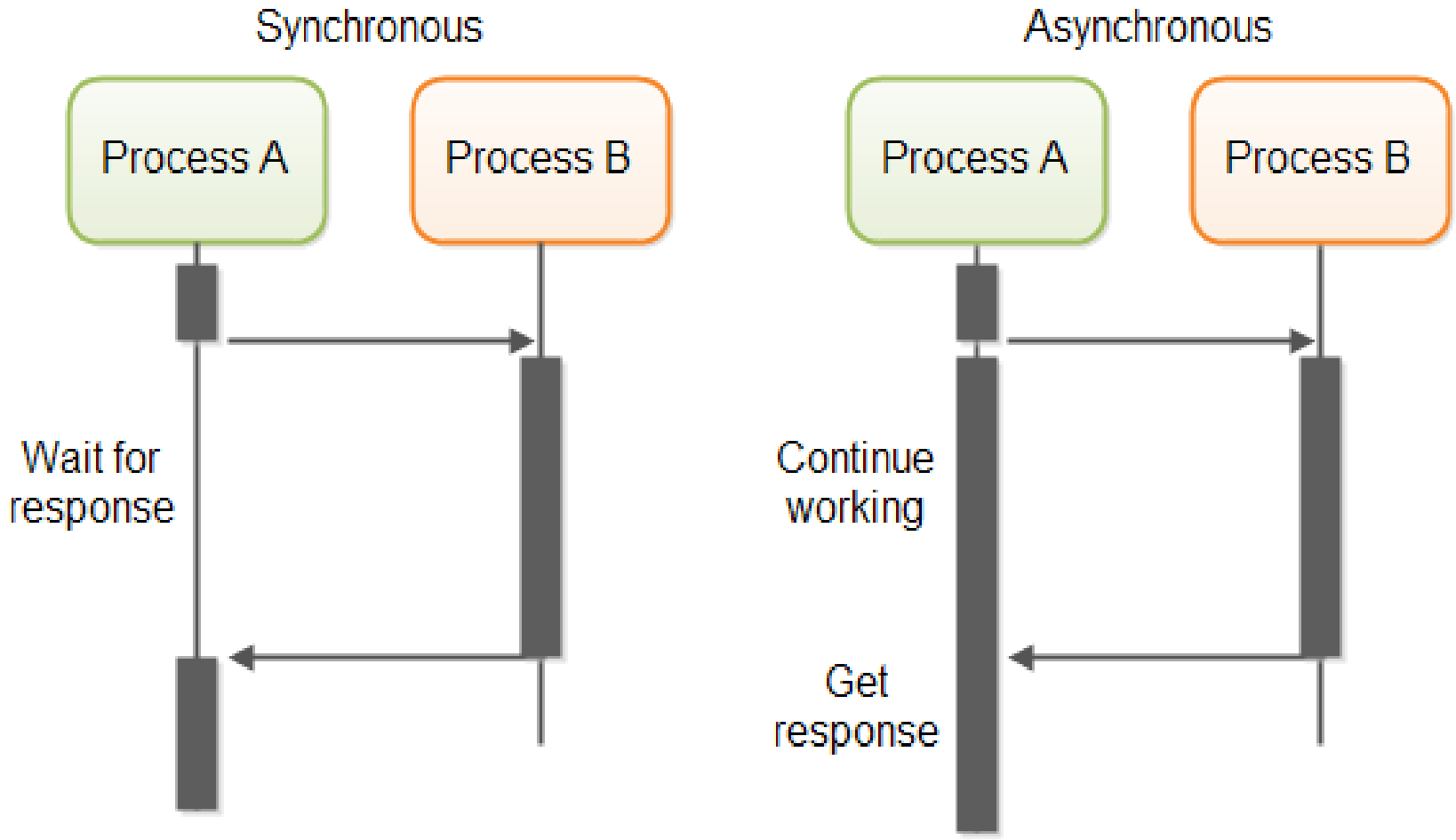
Komunikacja synchroniczna i asynchroniczna

Komunikacja:

Synchroniczna – w trakcie pobierania danych / zlecania zadań dla aplikacji użytkownik musi czekac na odpowiedz drugiej strony (servera, bazy danych).

Asynchroniczna - w trakcie pobierania danych / zlecania zadań aplikacji użytkownik może wykonywać inne czynności, może także pobierać dane jednocześnie z wielu miejsc i zlecać wiele zadań.

Przykład: technika tworzenia aplikacji internetowych AJAX (Asynchronous JavaScript and XML)


Synchronous
Process A
Process B
Wait for response
Asynchronous
Process A
Process B
Continue working
Get response

AGH

Biblioteka Boost::Asio

"...one of the most highly regarded and expertly designed C++ library projects in the world." — Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

Boost.Asio może być używana do operacji synchronicznych jak i asynchronicznych na obiektach I/O takich jak np. socket. Przed rozpoczęciem użytkowania Asio dobrze jest poznać koncepcje rożnych części biblioteki, aplikacje która piszesz i tego jak współpracują one między sobą.
Na początku rozważmy operacje poleczenia na gnieździe.

Będzie to operacja synchroniczna

Program będzie miał przynajmniej jeden obiekt io_service. Reprezentuje on połączenie programu z I/O systemu operacyjnego

```
boost::asio::io_service io_service;
```

Aby wykonywać operacje I/O program będzie potrzebował obiektu I/O takiego jak np. TCP socket

```
boost::asio::ip::tcp::socket socket(io_service);
```

1.Program inicjalizuje połączenie, wywołując obiekt I/O

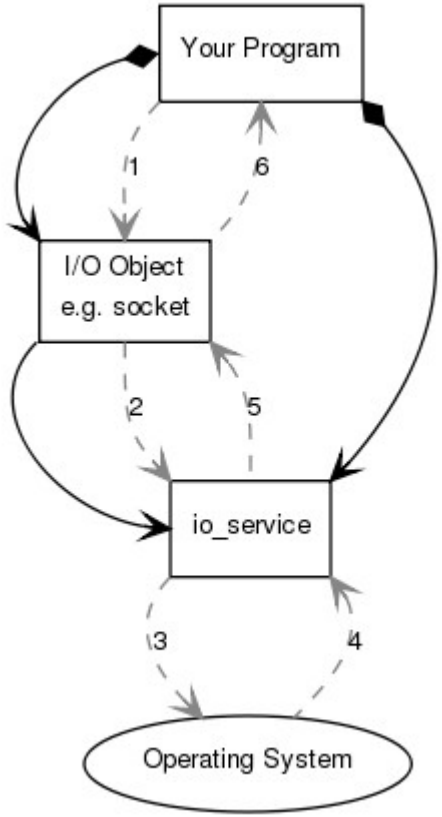
2. Obiekt I/O przekazuje zadanie do io_service

3. io_service wywołuje system operacyjny, aby wykonał operacje połączenia

4.System operacyjny zwraca wynik operacji do io_service

5. io_service tłumaczy wszelkie błędy
przekazywane przez OS na boost::system::eror_code. error_code może przyjmować rożne sprecyzowane wartości lub przyjmować wartość logiczną gdzie false oznacza brak błędów. /* error_code.txt */

6. Obiekt I/O rzuca wyjątki typu boost::system::system_error jeśli nie udało się wykonać operacji



Jeśli rozważamy operacje asynchroniczne, kolejne kroki wyglądają nieco inaczej.

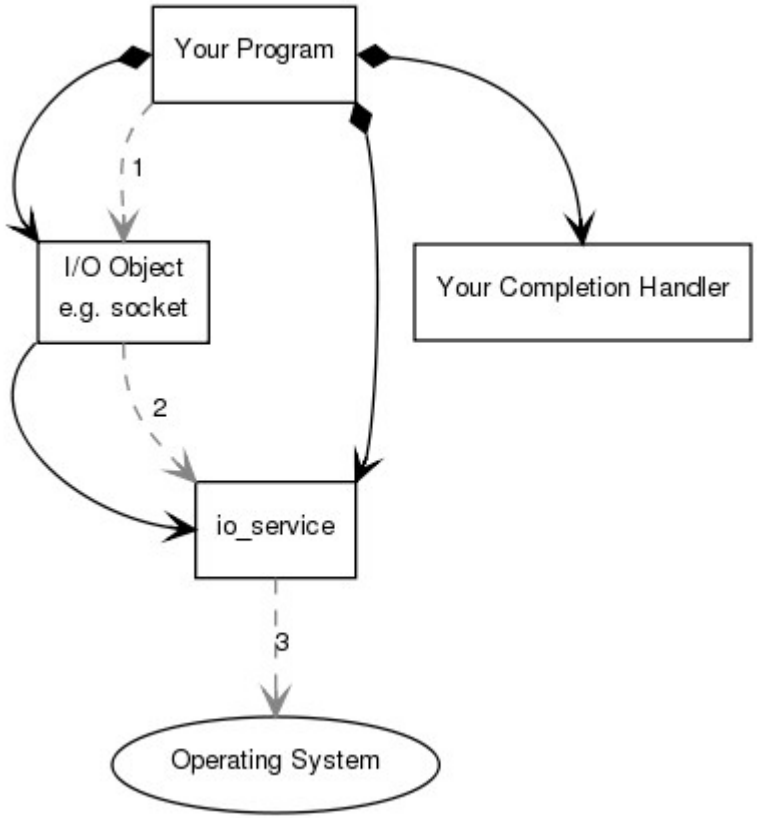
1.Program inicjuje połączenie poprzez wywołanie obiektu I/O

```
socket.async_connect(server_endpoint, your_completion_handler);
```

Jeśli funkcja *your_completion_handler ma następującą sygnaturę*

```
void your_completion_handler(const boost::system::error_code& ec);
```

2.Obiekt I/O przekazuje zadanie do io_service

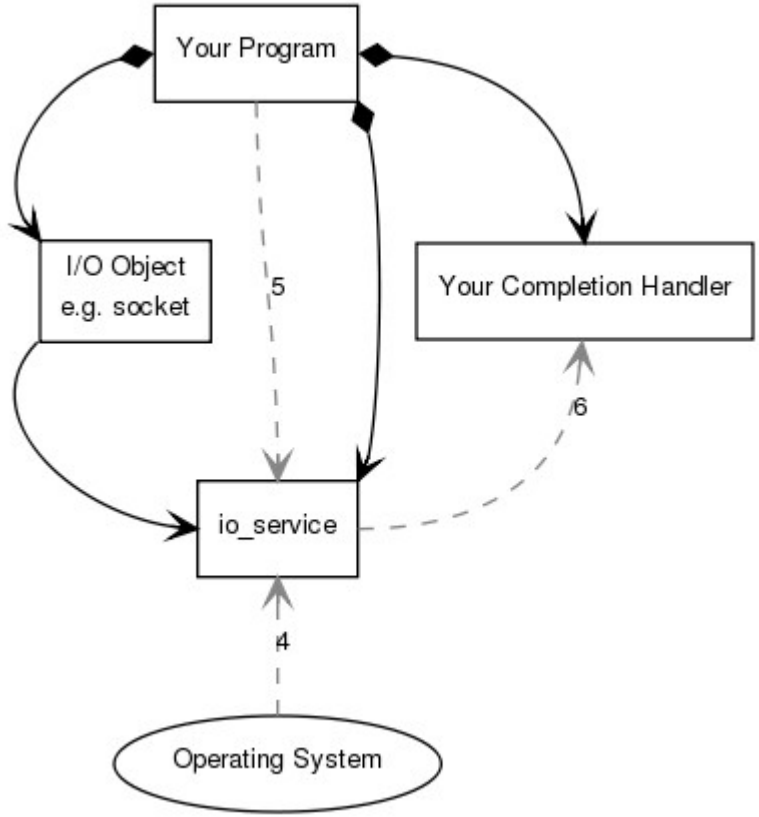


3. io_service sygnalizuje OS ze powinien rozpocząć połączenie asynchroniczne. Mija trochę czasu

4. OS sygnalizuje ze operacja połączenia została zakończona poprzez umieszczenia wyniku w kolejce, która obsługuje io_service.

5. Program musi wywołać funkcje io_service::run() aby wynik został zwrócony. Funkcja ta zatrzymuje się w momencie gdy jakieś operacje asynchroniczne nie zostały wykonane

6. Wewnątrz funkcji io_service::run(), io_service obsługuje kolejkę operacji, tłumacz je na error_code i wtedy przekazuje go do your completion handler



Bufory

Zasadniczo operacje I/O wiążą ze sobą transfery danych do i z ciągłych obszarów pamięci - zwanych buforami. Takie bufory można stworzyć za pomocą pary zawierającej wskaźnik na kontener danych i jego rozmiar w bajtach.

```
typedef std::pair<void*, std::size_t> mutable_buffer;
typedef std::pair<const void*, std::size_t> const_buffer;
```

Dzięki takiej reprezentacji, bardzo trudno doprowadzić do przepełnienia buforowanych danych, ponieważ użytkownik jest w stanie jedynie stworzyć drugi bufor w postaci jakiegoś kontenera który będzie miał rozmiar nie większy od narzuconego.

Strumienie, „Short write”, „Short read”

Wiele obiektów I/O w Boost.Asio działa na zasadzie strumienia. Oznacza to, że:

-nie ma granic wysyłanych danych, dane są transferowane w postaci ciągłego strumienia

- operacje otrzymywania i wysyłania danych mogą wysyłać mniejsze ich ilości niż się oczekuje

Uzywajac „wolnych” funkcji takich jak read(), write(), async_read(), async_write() oraz buforów jesteśmy pewni, że dany pakiet danych zostanie przeslany w całości.

TCP Client

Obiekt resolver pozwala na wyszukanie i połączenie w punkty dostępowe nazwy hosta oraz nazwy portu.

```
ip::tcp::resolver resolver(my_io_service);
ip::tcp::resolver::query query("www.boost.org", "http");
ip::tcp::resolver::iterator iter = resolver.resolve(query);
ip::tcp::resolver::iterator end; // End marker.
while (iter != end)
{
  ip::tcp::endpoint endpoint = *iter++;
  std::cout << endpoint << std::endl;
}
```

Lista otrzymanych punktów dostępowych może zawierać zarówno IPv4 jak i IPv6, więc program może próbować każdego z nich aż do momentu znalezienia tego właściwego
Aby się połączyć musimy stworzyć obiekt socket i wywołać funkcję connect().

```
ip::tcp::socket socket(my_io_service);
boost::asio::connect(socket, resolver.resolve(query));
```

```
boost::asio::async_connect(socket_, iter,
    boost::bind(&client::handle_connect, this,
      boost::asio::placeholders::error));

// ...

void handle_connect(const error_code& error)
{
  if (!error)
  {
    // Start read or write operations.
  }
  else
  {
    // Handle error.
  }
}
```

```
ip::tcp::socket socket(my_io_service);
socket.connect(endpoint);
```

Dane mogą być wysyłane lub odczytywane za pomoca metod:

receive(), async_receive(), send(), async_send()

Jednak bezpieczniej jest korzystac z:

read(), async_read(), write(), async_write()

Ponieważ pozbywamy się problemu short read i write.

UDP Client

Aby otrzymać listę punktów dostępowych używa się obiektu resolver

```
ip::udp::resolver resolver(my_io_service);
ip::udp::resolver::query query("localhost", "daytime");
ip::udp::resolver::iterator iter = resolver.resolve(query);
...
```

UDP socket jest przypisywany do lokalnego punktu dostępowego
Poniższy kod stworzy IPv4 UDP socket
i przypisze go do lokalnego hosta o porcie 12345

```
ip::udp::endpoint endpoint(ip::udp::v4(), 12345);
ip::udp::socket socket(my_io_service, endpoint);
```

Aby odczytywac lub zapisywac dane na polaczonym gniezdzie uzywamy funkcji:

receive(), async_receive(), send() or async_send()

TCP Servers

Program używa akceptora aby zaakceptowac przychodzące połączenia TCP

```
ip::tcp::acceptor acceptor(my_io_service, my_endpoint);
...
ip::tcp::socket socket(my_io_service);
acceptor.accept(socket);
```

Jeśli socket zostanie pomyślnie zaakceptowany, to server może czytać bądź wysyłać dane do clienta.

Liczniki (timers)

Długo trwające operacje I/O często maja określony czas działania (deadline timers).

Prosty licznik pelniacy funkcje oczekiwania synchronicznego:

```
io_service i;
...
deadline_timer t(i);
t.expires_from_now(boost::posix_time::seconds(5));
t.wait();
```

Oraz asynchronicznego:

```
void handler(boost::system::error_code ec) { ... }
...
io_service i;
...
deadline_timer t(i);
t.expires_from_now(boost::posix_time::milliseconds(400));
t.async_wait(handler);
...
i.run();
```

źródła:

- www.boost.org

- gynvael.coldwind.pl

- TCP/IP. Ksiega eksperta
  Karanjit S. Siyan, Tim Parker

- www.wikipedia.com

Dziękuje za uwagę :)